CONTINUOUS INTEGRATION USING LABVIEW, SVN AND HUDSON

O. O. Andreassen, A. Tarasenko, CERN, Geneva, Switzerland

Abstract

In the accelerator domain there is a need of integrating industrial devices and creating control and monitoring applications in an easy and yet structured way. The LabVIEW-RADE framework provides the method and tools to implement these requirements and also provides the essential integration of these applications into the CERN controls infrastructure. Building and distributing these core libraries for multiple platforms, e.g. Windows, Linux and OS X, and for different versions of LabVIEW, is a time consuming task that consist of repetitive and cumbersome work. All libraries have to be tested, commissioned and validated. Preparing one package for each variation takes almost a week to complete.

With the introduction of Subversion version control (SVN) and Hudson extensive continuous integration server (HCI) the process is now fully automated and a new distribution for all platforms is available within the hour. In this paper we are evaluating the pros and cons of using continuous integration, the time it took to get up and running and the added benefits such a solution has given to our team. We conclude with an evaluation of the framework based on the productivity and quality increase and finally indicate new areas of improvement and extension.

INTRODUCTION

Developing, building and distributing software at CERN is a mixed and challenging process: on one side, when working with operational equipment, it is mandatory to carefully plan potential impact on the accelerator complex, while on the other side, when working with experimental prototypes or test benches, new ideas and designs will be tested out all the time and the software has to be adapted quickly.

The LabVIEW Rapid Application Development Environment (RADE) [1] came to life to cope with this agile environment, giving users the means to quickly solve new challenges and at the same time provide stability for long-lived or critical applications. We have approximately 500+ LabVIEW users at CERN, of which ~100 uses RADE, all developing in their own unique environment. Therefore we had to create a release scheme that could be used in the most popular operating systems (Linux, Windows and OS X).

To ensure that bugs, requirements and other past experiences are considered in every new release, unit testing is performed on each critical item. This work and the associated release process itself where in the past all done manually or semi automated through scripts and custom tailored tools.

With the framework growth, a full-featured distribution typical would take from a day to a week to complete.

As an example one new RADE release cycle involves:

- Adding new libraries
- Running unit tests on the new libraries
- Validating the outcome of the test
- Bundle it all in to an installer.
- Testing the installer on a "clean" target (removing potential environmental misconfiguration issues)
- Rebuild, the application once deemed stable
- Uploading the release to the repository

This led us to investigate different automation and distribution methods such as Continuous Integration (CI), source controls and unit testing tools that were compatible with LabVIEW. The main challenge was finding a tool that could facilitate CI on a graphical application such as LabVIEW.

Trough studies and tests, several highly customizable and easy to use tools were identified, however either they did not offer any bindings or accessories facilitating integration of graphical programming languages, or they could not work in a cross platform environment which was another requirement for us.

All methodology, resources, scripts and deployment tools already used in the team were inventoried and assembled into a fully automated, integrated and low maintenance build engine that in less than one hour would test, build, document, distribute and deploy all our core LabVIEW libraries.

DEVELOPMENT METHODS

In order to reduce the testing and deployment time as much as possible while keeping the robustness from a traditional software project we landed on an agile based development style, with a test-driven execution. Through agile methods, tasks and projects are split into smaller increments that require minimal planning. Every iteration involves a small cross-functional team working on all disciplines: planning, requirement analysis, design, coding, unit testing and acceptance testing. At the end of the iteration, the product or result is demonstrated to the stakeholders, minimizing risks and giving room for fast changes and adaptations [2].

This methodology described by E. A. Edmonds in 1974 [1] became known later when a group of software developers published what they call the "Manifesto for Agile Software Development" [3].

CONTINUOUS INTEGRATION PRINCIPLES

CI is a software engineering practice where small or isolated changes are immediately tested and reported on when they are added to a larger code base. Therefore if a defect is introduced in the code base, it can be identified

ISBN 978-3-95450-139-7

and corrected without delay. In addition CI software tools can be used to automate testing and to automatically generate documentation [4].

Continuous integration has evolved since its conception. Originally, a daily build was the standard practice whereas the usual rule today is that each team member submits its work on a daily (or more frequent) basis and a build shall be conducted with each significant change.

Hence, when used properly, continuous integration provides constant feedback on the status of the software and its defects are detected early on in development. In addition and as side benefit the defects are typically smaller, less complex and easier to solve [4].

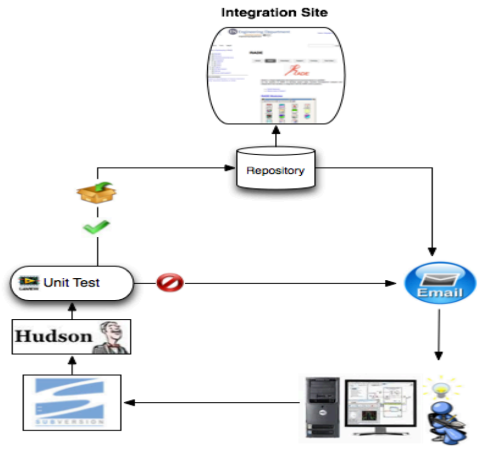

Figure 1: Continuous integration process.

Figure 1 shows a typical automated, test driven CI system, where developers commit their code to a central repository (SVN in this case), then the Hudson CI engine picks up the committed software (either “on change” or periodically), runs unit tests, and depending if the test(s) fail or pass, creates the deliverables and/or notifies the developer.

TOOLS SELECTION

An essential part of a test driven development environment is the tool selection. As CERN standardizes on SVN, it was a natural choice for source control, but the core element for complete automation is the CI engine that has to:

- Be compatible with the existing SVN repository
- Be able to execute any programming language or script needed in the build process.
- Run on all our main operating systems (Linux, Windows and OS X),
- Report any issue(s) encountered automatically.
- Be easy to maintain
- Have a plugin based and flexible pool of tools available

After evaluating the most common CI tools on the market we the selected the tool that had the easies setup and best interface: Hudson CI (table 1).

Table 1: Comparison of CI Tools

Name	Platform	SVN SCM support	Mail Support	Other Builders
Bamboo	Servlet	Yes	Yes	cmd line, Bash
Hudson	Servlet Container	Yes	Yes	Most scripting tools
CControl	Cross Platform	Yes	Yes	catch-all 'exec'
Continuum	JDK	Yes	Yes	----

FIRST INTEGRATION

A proof of concept was set up to make sure that the CI engine could repeatedly run the jobs needed without failing and creating false positives.

The conceptual test was performed on an SLC 5 x64 based machine. It involved the following steps:

- Download source from a SVN repository,
- Execute a LabVIEW application builder trough Hudson’s scripting interface
- Compile a simple test application.
- Running the application automatically trough the CI engine
- Write an output of the application to a log file and the console

Initial Testing, Issues and Solutions

The initial test was repeated every hour for 48 hours. Trough the test we made three important discoveries that had to be solved:

- On Linux it is not possible to build a LabVIEW based application without a graphical environment. We had to use a Virtual Network Computing (VNC) server [6] which would function as a frame buffer where LabVIEW could execute its graphical dependencies. The introduction of the VNC interface made it possible to run LabVIEW based server tools on headless Linux systems, and it mad it possible to graphically configure and intervene with server instances without the added development overhead of an additional client interface.
- The Hudson interface listens to standard input/output. A failed test will only be marked as failed if the application or script sets an exit flag not equal to zero. Whereas if you set an exit flag within the LabVIEW environment, it signals the application to quit and blocks any consecutive tests without restarting the application. This is solved by making use of the built in standard output pipes and then have a parallel Linux/windows batch process listening to the unit test outcome, setting an exit flag if it failed.

ISBN 978-3-95450-139-7

- The SVN authentication times out after 24 hours (using the ssh+svn protocol) and without any authentication management one has to manually log in after every timeout. This is solved by switching to “https” protocol and by using certificates for re-authentication [7].

Architecture

The RADE LabVIEW package has to be compiled for 5 different platforms, 32 and 64 bit operative systems and consist of many different build types. All the builds are orchestrated trough a main instance of Hudson, and built on 5 slave nodes. Figure 2 shows a simplified overview of the RADE CI architecture.

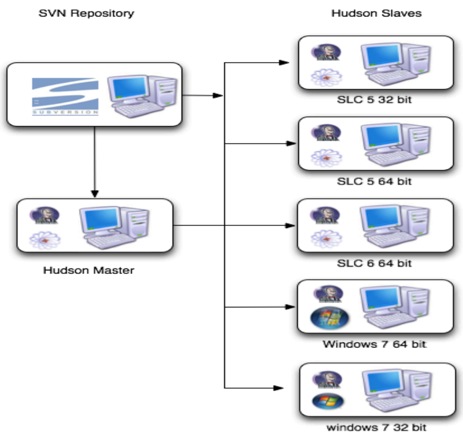

Figure 2: RADE CI architecture.

CI VERSUS MANUAL BUILDS

With the introduction of CI the release process is reduced from one day to about 1 hour (53 minutes), and makes it possible for the developers to work on other tasks while the build is taking place (no interaction needed).

In addition, automating the tasks removes typical “operator errors” that happen when doing repetitive work.

Hence it is now possible to introduce new toolkits and software in the framework in mere minutes and to distinguish between stable and unstable builds. As a result of this, we can support parallel releases for demanding and important customers, in the same environment used for the stable source.

An added bonus associated with the continuous integration is the early feedback. Since all new and legacy unit test are executed on every release, the developer get immediate feedback if a change broke modules in the framework, and can start sorting out the issues at once [5].

REMARKS AND ISSUES

However, automated builds have its downsides, to save time in the release and distribution process, you have to make compromises when setting up the environment:

- All stakeholders and developers have to follow precise guidelines, fixed naming schemes and structures not to break the automation, or cause trouble for other builds.
- The CERN passwords and certificate management imposes that a few times a year, the certificate enabling and authenticating communication with the source code repository has to change. This blocks the CI environment.
- To deal with a graphical tool such as LabVIEW is tricky in an environment designed for textual code. It implies setting up virtual displays, adapting fonts and avoiding dialogues in your builds that will block the execution.
- In the CI environment there isn’t a built in monitor or notification if one of the slave modules goes down.

CONCLUSION & FUTURE IMPROVEMENTS

The continuous integration and automation of the RADE framework has greatly improved the delivery time, quality and frequency of new software. It has made the framework more robust through preventive testing and fault elimination before distribution. Automating these tasks add some maintenance overhead for the build environment itself, nevertheless the advantages and overall time saved makes it worth the effort.

Since the slave nodes are not running build jobs all the time, we plan to service out the CI engine. By sending a simple command to the HCI interface, one can download, identify, build and deploy software on all targeted platforms in mere minutes.

We are also working on improving the overall build environment, centralizing the CI instances, making management of certificates, naming schemes and create more robust and less demanding templates.

REFERENCES

[1] O. Ø. Andreassen et al. “The LabVIEW RADE framework distributed architecture”, ICALEPCS (2011), Grenoble, France

[2] E. A. Edmonds, “A Process for the Development of Software for Nontechnical Users as an Adaptive System”, *General Systems* 19: 215–18.

[3] Beck, Kent et al. "Manifesto for Agile Software Development". Agile Alliance. Retrieved 2010-06-14. (2001)

[4] M. Rose, “Continuous Integration (CI)”, (2008), http://searchsoftwarequality.techtarget.com

[5] K. Beck, "Embracing Change with Extreme Programming". Computer 32 (10): 70–77 (1999)

[6] T. Richardson, Q. Stafford-Fraser, K. R. Wood, A. Hopper, "Virtual network computing", IEEE Internet Computing 2 (1998)

[7] S. Fisher, “Starting and accessing Hudson”, (2013), http://wiki.eclipse.org